LINC: A Compact Yet Powerful Coordination Environment

Maxime Louvel, François Pacull

LINC: a compact yet powerful coordination environment

Maxime Louvel, François Pacull

Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
17 rue des Martyrs 38000 Grenoble, France.
maxime.louvel@cea.fr ; francois.pacull@cea.fr

Abstract. This paper presents LINC, a coordination programming environment. It is an evolution of earlier middlewares (the Coordination Language Facility (CLF) and Stitch). The aim is to provide a more flexible and expressive language correcting several of their limitations and an improved run-time environment. LINC provides a compact yet powerful coordination language and an optimised run-time which executes rules. This paper describes the intrinsic properties brought by the LINC environment and how it helps the coordination aspects in a distributed system. This paper also emphasises on the reflexivity of LINC and its usage at system level. Finally, it illustrates through several case studies, how LINC can manage a wide range of application domains.

Keywords: Coordination, language, tuplespace, distributed systems

1 Introduction

Today's systems are not only distributed, they are composed of other systems more or less opaque. They have to interact with real world and thus have to consider on the one hand very small embedded systems and on the other end unbounded resources sprayed in the "cloud". Some pieces of work consider that the traditional approaches based on objects and services cannot hold such complexity [18]. In this context, coordination models and languages [25] are essential to coordinate basic elements as well as systems of systems. The last decades have seen a lot of work in the coordination area [24], starting with Linda [13]. Linda firstly introduced the notion of tuple-space as the ground for coordination. In Linda, components exchange and synchronise through tuples addition and removal in a shared tuple-space. This approach allows the decoupling of processes both in space and time. Indeed to exchange data between two components, the first one simply puts a tuple in the shared tuple-space. It does not have to worry if another component is currently waiting for this information or how this information should be exchanged. The data is exchanged when another component reads the tuple. The read may come before, at the same time or after the put.

Based on Linda, a number of evolutions have been proposed. Starting in the 2000s, researchers have focused on using tuple-spaces for mobile computing [8,12,

22, 33]. To support mobile environment, one of the main improvement is the use of several tuple-spaces instead of a single one shared by all the processes. Then, from mobile computing, researchers have focused on making applications context aware by the use of tuple-spaces [5, 7, 15]. To go a step further with the mobility, toward autonomous systems, researchers considered more intelligent tuples [21, 30]. They have shown the interest of relying on tuple-spaces in nowadays systems. However, there is still a gap between what people express and how the developer will implement it. We believe it is essential to provide means for developers to focus on the coordination of the systems. It is also tremendously important to provide them with a simple programming environment powerful enough to handle the system complexity. The response to the management of complex system should not be a complex programming environment.

This paper presents the full set of LINC features with a special focus on how it eases the coordination tasks. It complements partial description in earlier application domain oriented papers [10,20]. LINC is a compact yet powerful coordination environment which relies on the three basic primitives of Linda to read, add and remove tuples in a tuple-space. It uses distributed tuple-spaces called bags. A bag is responsible for storing the tuples and may provide a special implementation of the three primitives. This is very convenient when communicating with the physical world (e.g. sensors or actuators) or when integrating legacy systems. LINC uses production rules to interact on the tuple-space rather than using imperative code to glue the primitives. Actions on the bags are embedded into distributed transactions which simplifies a lot the job of the developer that does not have to worry about writing code to roll back half of the actions done so far when something goes wrong. Transactions also enforce the consistency between the actual system and the software view. This, once again aims at helping developers to focus only on the coordination.

The paper is structured as follows. Section 2 presents the coordination language of LINC. Then, Section 3 describes its main features. Section 4 sketches several concrete case studies where LINC has been used. Section 5 positions our approach with respect to related works. Finally, Section 6 concludes the paper.

2 LINC coordination language

LINC is the natural evolution of the Coordination Language Facility (CLF) [2] and Stitch [1] middlewares initially developed for deploying distributed applications. The three of them share the resource oriented approach manipulated through a high level rule-based language. However, the architecture of LINC has been completely revisited to adapt to the new landscape defined by the combination of the Internet-of-things, the cloud and the Cyber-Physical Systems.
The main differences are:

- in LINC, the coordination engine has been improved both in term of CPU usage and memory footprint. It is embedded in every object while CLF/Stitch relied on more complex dedicated objects. This allows to better distribute the coordination by delegating some parts to more modest CPU;

– the rule language has been extended to improve its expressiveness;
– the environment comes with tools: monitoring, rule analysis (memory, time and dependencies) and a replay mechanism allowing post-mortem re-execution preserving causal order to debug the initial run. Tools are not the focus of this paper, details may be found in the LINC wiki (http://linc.middlewares.info).

2.1 LINC roots

We briefly recall here the basics of LINC to make the paper self contained. LINC, like CLF or Stitch, is rooted in Linda-like tuple-spaces (*Associative Memory*), *Production Rules* and *Distributed Transactions*.

Associative Memory: The global tuple-space is composed of several distributed tuple-spaces called bags. Tuples in bags are accessed with the three operations: `rd()`, `get()` and `put()`. The `rd()` verifies the presence of resources matching some given criteria: resources corresponding to the requested pattern (partially instantiated tuple) passed as parameter. The `get()` (`in` in Linda) removes a tuple while the `put()` (`out` in Linda) inserts a new tuple in a bag.

Production Rules: In LINC, the `rd()`, `get()` and `put()` primitives are invoked through production rules [9]. This prevents to write a huge amount of imperative code such as Java, C or Python by the use of a coordination language to define how the resources are manipulated in the system. A production rule is decomposed in a *precondition* and a *performance* part. The precondition part uses `rd()` operations combined with an inference engine in order to evaluate distributed conditions in the system. Then, the performance part uses:

– `rd()` to verify that conditions are still valid at performance time;
– `get()` to consume resources (e.g. manage critical resources, consume events);
– `put()` to generate resources (e.g update the system, command actuator).

A rule typically uses several bags, physically distributed or not, to access the necessary information and update the system accordingly. The particularity of LINC is that the performance part is embedded in Distributed Transactions.

Distributed Transactions: They are used in the performance part to ensure the all-or-nothing [6] property. They group in the same set of operations the verification of some conditions, the consumption of critical resources and the update of the global state of the system. Thus, the performance part is a list of transactions that are executed in sequential order.

The combination of these three paradigms enables to build transactional reactions to complex events. Complex events in a building automation context could be the combination of several events such as *people in the room* and *temperature lower than sixteen degrees* and *HVAC system is OFF*. Transactional reactions consist in *put the heating ON*, and *update the state of the system*.

2.2 Bags abstraction

The main interest of splitting the global tuple-space into several bags is that each bag may define its own semantic associated to the `rd`, `get` and `put` and

the tuples themselves. As a result, bags can encapsulate software or hardware components.

a database: each table of the database can be associated to a bag; `rd()` and `put()` corresponds to the `reads` and `writes` on the database.

a service: This concerns remote services as well as local services directly embedded in the bag. The partially instantiated tuple composed of the input parameters is passed to the `rd()`. The concerned service is invoked from the bag, using the legacy protocol imposed by the remote service (e.g. SOAP for a Webservice based approach or a native Remote Procedure Call). The output parameters obtained as the result of the service is added to the input parameters defining the fully instantiated tuple returned by the `rd()`.

an event system: Tuples contain the topic, the ID, the timestamp and a payload; the `rd()` and `put()` operations correspond to `subscribe` and `publish` of the event system.

a sensor: Data are collected from the gateway and inserted into different bags storing the various relevant information. For instance, for a very simple approach, we can consider the three bags associating the `id` of the sensor to the sensed `value`, the `type` of sensor and its `location`. Successive values can be obtained through the `rd()`, changing the location of the sensor is as simple as modifying the resource by consuming it with a `get()` and inserting the updated information with a `put()`. The sampling frequency, the type of bag (e.g. set, multiset, FIFO, ...) or the precision can be adapted at this stage. The behaviour of the `rd()` may also be adapted to replace a simple tuple matching by a more complex matching: e.g. interval, fuzzy logic or ontologies.

an actuator: When interacting with the real world, the `put` operation is typically used to send actuation commands (e.g to set the speed of a motor, the direction of the wheels or to power on or off a subsystem). We can act on the physical actuator through a `put()` operation with a resource with the `id`, the `command` to be applied and the possible parameters `p1`, `p2` into the associated bag. This is enough to trigger action to the actual actuator.

Finally, it is possible to associate bags. For instance, one bag can contain the number of resources contained in the bag it is associated to. More complex associations can be considered such as arithmetic functions (e.g. sum, average, max or min or even a Bayesian filter). The main advantage is to have a direct access to refined information computed from a set of resources. It has been used for instance in the application described in 4.2 to filter outliers values coming from a matrix of Rfid readers.

2.3 Coordination language

Bags are grouped into *objects* for identification purposes, thus objects are a logical decomposition of an application. For instance an object may manage all the sensors communicating with the same protocol or located in the same space. An object may execute rules to coordinate the system by acting on its own bags and the other objects' ones. When an object executes rules, it plays the role of coordinator. Rules can be executed by any object. This means that

the programmer is free to distribute them among the different objects of an application.

To illustrate how LINC rules are working Listing 1.1 gives a very simple rule involving two sensors (presence and temperature) and an actuator (heating controller) using three different technologies (e.g. different protocols). This rule adjusts the heating of a room when someone is inside and the temperature is lower than 16 °C. The rule is composed of the *precondition* (when to trigger the rule) and the *performance* (what to do) separated by the symbol `::`.

```
{*,!}["Techno1", "sensors"].rd("pres_1", "True") &                          1
{*,!}["Techno2", "sensors"].rd("temperature_a", temp) &
ASSERT: temp < 16 &                                                         3
{1,!}["Techno1", "location"].rd("pres_1", location) &
{*,!}["Techno3", "actuators_list"].rd(id_act, "heating", location)          5
::
{                                                                           7
  ["Techno1", "sensors"].rd("pres_1", "True");
  ["Techno2", "sensors"].rd("temperature_a", temp);                         9
  ["Techno3", "actuators"].put(id_act, "heating","2");
}                                                                           11
{ ... #other group of actions }.
```

Listing 1.1. LINC rule example

Precondition The precondition is composed of *tokens* processed by an inference engine with a right propagation. Listing 1.1 contains 8 tokens (1 per line). The first token invokes a `rd()` operation into the bag *"sensors"* of the object *"Techno1"*. This object encapsulates the gateway of the first technology. The token looks for the resource *("pres_1", "True")* where the first field is the sensor id (generally imposed by the technology) and the second the value *"True"* meaning that a presence is detected. In a similar way, the second token asks for the temperature currently sensed by the sensor whose id is *"temperature_a"*. When the `rd` is done, all the matching tuples are returned one by one. For every returned tuple, a new branch is created with the value of the instantiated variables `temp`. The instantiated variables are right propagated. For instance in the third token this variable is compared to the threshold (16 degrees) thanks to the `ASSERT:` extension[1]. If the condition is false, the rule will not progress. If it is true, the next token asks the bag `location`, of the object `"Techno1"` (responsible for the presence sensor), for the location of sensor `"pres_1"`. With this information, the last token can ask the *id* of the co-located heating system.

The tokens in the precondition phase are preceded with modifiers embraced in curly brackets. The first field defines the number of maximum expected replies awaited by the `rd()` operation. Normally a `rd()` is blocked waiting for new matching resources that could become available. In this example, we use `*` for the first 2 tokens since we want to have all the replies. On the contrary, for the 4th we used `1` since a sensor has a single location and it is useless to wait for another resource. The second field is used to define the amount of time a pending `rd()` is waiting for a reply. Both are used to reduce the size of the inference tree as detailed in Section 3.2.

[1] The extensions `ASSERT:`, `COMPUTE:` and `SLEEP:` allows to respectively verify a condition, execute simple computation or wait for a given time.

Performance The performance part may contain several transactions enclosed in curly brackets. The first transaction embeds the verification of the conditions presence and temperature and the operation on the heating system. Indeed, if the presence detector becomes false, this means the person is out of the room and nothing is required from the system. If the temperature has changed, it is not required to react. Indeed another inference will start with the new temperature, and a more accurate heating command will be computed. Finally, the last action of the transaction does the required operation on the heating system. Contrary to the Event-Condition-Action [32] model, events and conditions are managed in the same way. Moreover, they may be enclosed with actions in a transaction.

The transactions are executed sequentially. If several transactions perform a `get()` on a same unique resource, they become exclusive. It is explained in context with the guards controlled alternatives (Section 3.3) and the graceful degradation (Section 3.4) mechanisms. The distributed transactions are enforced through a classical 2 Phase-Commit [6] to provide atomicity property (all-or-nothing). However, no hypothesis is made on how the bags implement these 2 phases. Thus, it is possible on the one hand to take advantage of the context and to implement a bag fully compliant with the 2PC on small microcontroller if required. On the other hand, it allows for a given bag to relax the 2PC.

2.4 Improvement with respect to CLF/Stitch

The explicit usage of the operations `rd()`, `get()` and `put()` in LINC allows to verify in the performance phase only the useful resources while in CLF/Stitch all the resources used in the precondition where systematically verified in the performance phase. The major drawback is that some `rd()` were verified even if it was not required. Indeed, among the `rd()` operations done in the precondition a significant part are just informative and not subject to change. Verifying them again in the performance was a waste of time and resources.

Another difference concerns the `rd()` and `get()` done in the performance. The presence of the resources returned in the precondition is verified in CLF / Stitch via a unique resource identifier. In LINC, it is not based on this identifier but on the value. This decreases the size of the inference tree and the amount of useless work when we have to deal with multiple resources with the same value (e.g. a sensor returning the same value).

Moreover, CLF considers in the performance a transaction for the `rd()` and the `get()` operations and the guaranty that the `put()` operations are eventually done by re-trying them until completion. This was motivated by the fact a `put()` was not supposed to fail. Indeed, for a tuple space, it is reasonable to consider that inserting a resource is always possible. However, in LINC we want also to target physical world, such as actuators that may fail when a `put()` is tried.
Thus, LINC enforces a transaction for (`rd()`, `get()` and `put()`) operations allowing a richer transactional model in the performance part.

In addition to a stronger model, we do not restrict the performance to a single transaction but we can have a sequence of transactions. This brings the possibility of alternative treatments sharing the same precondition part. This

not only decreases the work to be done by the coordination engine but also offers a better view to the programmer by replacing a set of CLF/Stitch rules by a single LINC rule. Powerful mechanisms using this capability, such as guards controlled alternatives and graceful degradation are described in Section 3.

Finally, the usage of modifiers in the precondition offers the programmer to specify information that helps the inference engine to better optimise the size of its data structures. This reduces both memory footprint and CPU usage.

All these improvements brought the required scale down in the rule management that allows LINC to target smaller computing units such as Raspberry PI, Pandaboard or even ARM9 custom board.

3 LINC features

This section presents several features offered by the LINC coordination language.

3.1 Control the frequency of a rule

In some circumstances it can be useful to control the pace of a rule. For instance, in Listing 1.1 the rule is triggered each time the temperature changes and a presence is detected. This is obviously too much for controlling the heating system. We can modify the rule by adding a new token at the first place of the precondition and at the first place of the performance part (cf. Listing 1.2).

We use a bag of type `set` called `tick` in which a new instance of the same resource is regularly inserted. The set property guaranties that a resource with the same value is present only once at a given time even if inserted several times (e.g. every 10 minutes). The precondition waits for the presence of this resource to start the evaluation of subsequent `rd` operations. If, in addition a token doing a `get` is added in the performance, it enforces that only one performance of the rule is performed per tick. Indeed, even if several instances of the rule reach the performance point, only one transaction will succeed. For the others, the `get` action (e.g. in line 5) will fail because the resource is not in the bag anymore.

```
{*,!}["Control", "tick"].rd(tick) &
... #tokens of Listing 1.1                                    2
::
{                                                             4
  ["Control", "tick"].get(tick);
  ... #tokens of Listing 1.1                                  6
}
```

Listing 1.2. LINC rule with tick

3.2 Reduction of the inference tree

When a rule is executed, an inference tree is built. Its size, and the number of branches waiting for new resources to appear may become a problem, especially on embedded devices. Hence, it is important to limit the size of this inference tree to limit the memory used and to decrease the CPU load. For this, LINC relies on directives exploiting the knowledge of the developer and on an automatic process (garbage collector pruning useless branches).

Information from the developer: Via the modifiers introduced in Section 2, the developer can:

Limit the number of matching tuples to consider is typically used when the developer knows that only a given number of resources is really useful. For instance in the rule of Listing 1.1, the 4th token asks for the presence sensor location. As a sensor has a single location, we know that we can close the flow of reply to the `rd()` operation right after the first reply. This can be done by replacing the `*` in {`*`,`!`} with `1`, i.e. the number of expected replies.

Limit the time to wait for matching tuples is used to model the expiration of some tuples. For instance, in the modified rule of Listing 1.2 a tick is generated every 10 minutes. A new branch is then started at each generation of a new tick resource. Thus, if no presence is detected during 2 hours, 12 branches will be waiting for the presence detection. When a presence is detected, the 12 branches will be activated, creating useless work. If we replace the `!` in {`*`,`!`} with `600`, the number of seconds to wait, the `rd("pres_1","True")` operation will be closed after 10 minutes and thus only one branch is active at a time.

Garbage collector When waiting on `rd` for matching tuples, it might become a point where branches of the inference tree built so far do not make sense any more. For instance here, when the presence sensor becomes `"False"`, the resource *("pres_1","True")* disappears from the bag (i.e. it is replaced by *("pres_1","False")*). Then, it is not necessary to continue to maintain the branches depending on *("pres_1","True")*. Indeed when the performance will be executed it will fail because this resource is not available.

A garbage collector periodically browses the inference tree asking the bags if the tuples are still present. By tuple we do not mean the exact same tuple but one with the same value. If the tuple is not there, the branch of inference tree starting from this node can safely be garbaged. Indeed, continuing the precondition will only lead to failures in the performance phase. If a branch is garbaged and the same value is added again in the bag, this will trigger a new inference.

3.3 Guards controlled alternatives

```
...                                                                    1
::
{                                                                      3
  ["Techno1", "sensors"].rd("pres_1", "True") ; #First guard
  ["Techno2", "sensors"].rd("temperature_a", temp);                    5
  ["Techno3", "actuators"].put(id_act, "heating","2");
}.                                                                     7
{
  ["Techno1", "sensors"].rd("pres_1", "False") ; #Second guard         9
  ["Techno2", "sensors"].rd("temperature_a", temp);
  ["Techno3", "actuators"].put(id_act, "heating","1");                 11
}.
```

Listing 1.3. Extension of rule in Listing 1.1 with guards

In the rule of Listing 1.3 we show how to implement a simple guards mechanism in the performance. Here we define how to manage the heating depending on the occupant's presence (i.e. putting the set point to *"1"* or *"2"*). Here, we do not test the presence in the precondition since we want to act in both case. We use 2 transactions, one for each case and we place in each of them a `rd()` respectively on *("pres_1","True")* and *("pres_1","False")*. Depending on the actual value of the resource, one of them commits and the other aborts.

3.4 Graceful degradation

Graceful Degradation is achieved by adding in the transactions a `get()` on a unique resource (created by the first transaction at line 3 in Listing 1.4). As transactions are tried in sequence, if transaction A succeeds, i.e. the heating command is successfully done, transaction B fails at line 9. If the heating system is not reachable, transaction A fails, transaction B succeeds (the temperature and presence have not changed). The `unique` is consumed and a SMS is sent to alert the maintenance. If the temperature has changed (or nobody is in the room anymore) at performance time, transactions A and B fail. However, as a new temperature resource is now available it is taken into account by another instance of the rule. Thus, a single rule can define what to do in the normal case and what to do in case of partial failures.

```
...
::                                                                          2
{["Control", "unique"].put(unique);} #create unique
{ #transaction A                                                            4
  ["Control", "unique"].get(unique);
  ...                                                                       6
}
{ #transaction B                                                            8
  ["Control", "unique"].get(unique);
  ["Techno1", "sensors"].rd("pres_1", "True");                              10
  ["Techno2", "sensors"].rd("temperature_a", temp);
  ["Alert", "SMS"].put("512123123", "Heating system is broken");            12
}
{["Control", "unique"].get(unique);}. #garbage unique                       14
```

Listing 1.4. Extension of rule in Listing 1.1 with graceful Degradation

3.5 Mutual exclusion

Mutual exclusion is not easy to solve with classic programming schemes such as semaphores or monitors. In LINC, the transactions greatly simplify the problem. To illustrate this, we propose our implementation of the classical philosophers dinner. The problem is solved by the two rules of Listing 1.5.

The bag `philosopher` contains resources associating the philosopher name and the id of his left and right forks. The bag `fork` manages the critical resources: the forks. When the first rule succeeds, it means a philosopher has gotten its two forks and moved from thinking to eating. If another rule instance wants to get a fork already attributed, it will fail because the resource is not in the bag anymore. The second rule makes the philosophers go from eating to thinking and put back the fork resources in the bag. This will wake up instances of the first rule waiting for available forks.

```
# thinking -> eating
{*,!}["Dinner","philosopher"].rd(name, fork_l, fork_r) &
{*,!}["Dinner","fork"].rd(fork_l)&
{*,!}["Dinner","fork"].rd(fork_r)&
{*,!}["Dinner","state"].rd(name, "thinking") &
::
{
  ["Dinner","fork"].get(fork_l);
  ["Dinner","fork"].get(fork_r);
  ["Dinner","state"].get(name, "thinking");
  ["Dinner","state"].put(name, "eating");
}.
# eating -> thinking
{*,!}["Dinner","state"].rd(name, "eating")&
{*,!}["Dinner","philosopher"].rd(name, fork_l, fork_r) &
SLEEP: 10
::
{
  ["Dinner","state"].get(name, "eating");
  ["Dinner","state"].put(name, "thinking");
  ["Dinner","fork"].put(fork_l);
  ["Dinner","fork"].put(fork_r);
}.
```

Listing 1.5. Philosophers dinner problem in LINC

3.6 Rules activation / deactivation

One major issue when dealing with rules is how to control them and to ensure some guaranties when we decide to enable or disable some of them. To control the rules execution, we rely on the reflexivity of LINC. Indeed, in LINC everything is a resource in a bag. Naturally, rules are also controlled by resources in a dedicated bag of coordinator objects. This bag is called `RulesId` and contains tuples shaped as *"(rule_id, status)"*. When the rule is compiled, a `rd` is added at the beginning of the precondition and in each transaction as shown in Listing 1.6. The variable `ego`, used for object name, refers to the object executing the rule.

```
    [ego, "RulesId"].rd("RU0001", "ENABLED");
```

Listing 1.6. Control of rules execution

Adding these `rd` allows to stop a rule by simply changing the resource *("RU0001", "ENABLED")* to *("RU0001", "DISABLED")*. Indeed the `rd` on the rule status will make the transactions fail. To reactivate the rule, the resource status is put back to *"ENABLED"*. Note that the `rd` on the rule status is the first `rd` of the precondition. This has three main interests:

- when disabled, no new inference is started;
- when disabled, the inference tree is completely garbaged because the resource *(rule_id, "ENABLED")* is no longer in the bag;
- if *(rule_id, "ENABLED")* is put back, a new inference tree is built.

Note that this generic principle used at system level can be easily used at application level to activate and deactivate groups of rules according to application context. For instance, we can use the same principle with a resource controlling a set of rules. This can be used, for instance, to put in place a very sophisticated scenario manager in the building automation domain [10].

3.7 Dynamic rules generation

For some applications, it can be required to dynamically generate rules corresponding to contextual information. To do so, a resource is added in a dedicated bag, called `AddRules`. This bag receives resources of the form *(package,source)* where `package` is the logical name of a group of rules and *source* the actual code of the rules (as shown in Listing 1.1 for instance).

When a resource is added in this bag, the rule is dynamically compiled. This compilation includes syntax verifications, and various checks to prevent potential issues at execution time. For instance the coordinator object checks that the bags used by the rule are accessible, and that each variable will be instantiated at some point by a `rd()` operation in the precondition phase. If the rule contains no detectable error, the coordinator starts to execute it right away.

3.8 Registry-based programming

When interfacing with the hardware (very small embedded systems such as complex actuators) it is required to prepare the data in some dedicated registries and then to trigger the global action by acting on the control register. To reproduce this behaviour in a rule is straightforward because actions inside a transaction are executed sequentially. As shown in Listing 1.7, where two bags are used, one to map the data registers and one to map the control one. It is just mandatory to place the action of the control register at the last position in the transaction.

```
...                                                                    1
::
{                                                                      3
  ["object", "data"].put("r1", value1);
  ["object", "data"].put("r2", value2);                                5
  ["object", "cmd"].put("send");
}.                                                                     7
```

Listing 1.7. Registry like programming

4 Case studies

This section presents case studies where the features of LINC have been used.

4.1 Building automation

Building automation is a typical case where coordination is essential and may become very complex. We need to coordinate a high number of sensors (e.g. temperature, light, co2, presence) and actuators (e.g. Heating, Ventilation and Air-Conditioning (HVAC), dimmable spotlights). These devices belong to independent subsystems distributed in the building and using different protocols: BACnet [3], LONWorks [19], KNX [16] or Zigbee, 433Mhz/868Mhz for wireless. These subsystems work autonomously but most of the time cannot cooperate.

In the context of the SCUBA (Self-organising, Co-operative and robUst Building Automation) FP7 project [28], LINC has successfully been used to offer the abstraction layer required to make all these devices able to coexist. In

addition LINC has provided the coordination in order to allow the binding of devices of different constructors across a set of buildings [10]. Several scenarios have been defined to coordinate the HVAC and the lighting systems in order to improve the energy efficiency. The graceful degradation feature of LINC has proven very efficient to handle partial failure of autonomous systems. The application is currently distributed across 6 partners' sites controlling 5 buildings.

Another important role played by LINC was the administration of autonomous subsystems according to context. It has been used to reconfigure the LONWorks bindings in a room that can be either two individual offices or a single larger room, depending on the presence of a removable wall. The binding of the buttons, temperature sensors, motion detectors to the lights, shutter, HVAC have to be reconfigured accordingly. In LONWorks, this would involve the manual intervention of a skilled technician. With LINC we have coordinated the reconfiguration process with rules dynamically generated according to the current context [29].

4.2 RFID table

To illustrate the capability of LINC to manage complex events detection, [20] describes our experiment with an original hardware. This hardware is a table stacking a 42" LCD screen with a HD 1080p resolution on top of a set of rfid readers organised as a matrix of 6 x 4 tiles, with each tile containing itself a matrix of 4 x 4 rfid readers. As a result there are 24 x 16 (384) rfid readers distributed in the table. The table works with classical rfid tags that can be attached to any physical object. The raw information received is, for each rfid reader, the set of detected tags. In addition, we have encapsulated as LINC bags two software components: a 2D engine able to display arbitrary content on the screen table allowing user interaction and a 3D engine able to render in a virtual world the tagged physical objects placed on the table.

The coordination language of LINC allowed to manage complex events resulting from the manipulation of the tags frequently added and removed from the table. The full application is managed by a dozen of rules distributed on the laptop responsible for displaying the 3D scene on the external display and the Raspberry PI managing the screen embedded in the table.

4.3 Smart actuators

In [14] we have designed smart actuators able to directly understand the LINC coordination protocol and thus to be participant to transactions. The actuator is thus able to locally detect that it will not be able to do the requested action. For instance, this may be due to currently insufficient energy or unfeasible physical actuation (e.g. due to an obstacle or an out of range request). This simplifies the error management and allows synchronised physical actions. For instance, we used these smart devices for the obstacle avoidance of an autonomous robot where actions on the motors failed when associated sensors detect obstacles in the considered direction. The bags encapsulating the control of the motor have been implemented on small microcontrollers of type ATmega328 or PIC24.

5 Related works

Since the introduction of tuple-spaces by Linda [13] in 1985, many contributions have been proposed to improve, extend and adapt the model.

MARS [8] extends the Linda tuple-spaces to add reactions. A reaction is implemented by an agent which triggers an operation on a matching tuple. Reactions are implemented with "meta" tuples containing a reference to the agent. In LINC, the reactions are defined by high-level coordination rules which embed actions in distributed transactions.

LIME [23] (Linda In a Mobile Environment) replaces the globally accessible persistent tuple-space of Linda by transiently shared tuple-spaces. In LIME each agent has its own tuple-space. When agents meets, they form a shared tuple-space and can exchange tuples. Strong reactions (ensuring a transaction) are restricted to a host or an agent. For distributed reactions, Weak reactions are used to ensure that eventually the reactions will be done if connectivity is preserved. LINC always uses distributed transactions in order to maintain the system in a consistent state. A similar approach to LIME is proposed in [12], with a lighter implementation of tuple-spaces.

The Holoparadigm [5] is a programming model to build context aware applications which introduces the concept of *Being* containing an interface, a behaviour and a history. History is a blackboard (similar to tuple-spaces) with Linda-like primitives. Holoparadigm offers the architecture for building agents however it mixes the coordination with the agents' code.

EgoSpaces [15] is a middleware targeting development of context aware mobile applications. It defines an agent as a unit of modularity and mobility with its own local tuple-space. EgoSpaces adds the *View* concept to limit the data seen by an agent (e.g. cost of the communication, physical location, thresholds on data). The view concept is interesting because it allows the developer to define when an agent should react.

The MobiGATE Coordination Language (MCL) [33] insists on the separation between computation and coordination which is a shared approach with LINC. Their approach seems well suited for distributing a stream or a known service in a mobile environment. However, in MCL it seems difficult to focus on the coordination when a very complex and dynamic system is considered. UbiCoMo [7] proposes a coordination model that mainly focuses on accessing data in ubiquitous environments. However, this limits too much UbiCoMo expressiveness to fit into this specific paradigm. MCL and UbiCoMo share with LINC the importance of separation between coordination and computation.

In [11] the authors propose a process-based methodology to design event-based mobile applications. They aim to translate UML activity diagram to event based models offering more flexibility. As outlined in their conclusion, event-based approaches does not make synchronisation easy. We believe this issue could be overcome by the transactional guaranties of LINC. We also believe that LINC rules could be generated from the activity diagram. This could be an interesting track to consider.

In [17] the authors present a programming model for concurrent coordination patterns targeting highly parallel and distributed applications. Similarly to LINC, they provide a language to focus on the coordination in complex environment. They aim to provide a high level language, relying on a formal language from which LINC could also take advantage of. Finally, their model relies on a tuple-space middleware. LINC seems a good candidate to implement their model. Generating LINC rules from a higher level model is in our future work.

In TOTA (Tuples On The Air) [21], tuples are not associated to a specific host, they are injected in the network and can autonomously propagate in the network according to a specified pattern. A tuple is defined by a content and a propagation rule. The TOTA approach might be interesting in some context where the data moves around the network while hosts come and go. However, in other cases this might include an overhead in the traffic because tuples are propagated even if no host is interested in them. On the contrary, in LINC, tuples are exchanged only when a rule is needing them. We believe that the autonomous and self evolution idea of TOTA could be implemented in LINC thanks to its reflexivity. Indeed objects and rules can evolve to reach an emergent behaviour.

Inspired by Gamma [4], another approach to reach self evolving system is to use chemical inspired tuple-spaces [30]. The idea is to rely on the semantic of chemical reactions to build coordination laws. High level information are then given on tuples and they autonomously evolve and move. The goal is to have an emergent behaviour with this approach. For instance, services scarcely used will automatically disappear from the tuple-space. This work only focus on the long run emerging behaviour, they do not allow to directly build coordination for the physical world. Moreover, focus has been on simulation [26] even if a middleware is under development in the scope of the SAPERE project [27]. Finally, in [31] the author define a spatial computing coordination language to extend Linda with space and time information in the tuples. As far as we understand, no implementation exists so far. LINC, with its reflexivity, could implement these proposal on large scale distributed systems.

6 Conclusion

This paper has presented the LINC coordination environment. It provides a compact yet powerful coordination language based on the three paradigms: associative memory, production rules and distributed transactions. This combination provides a language with a high expressiveness. High level coordination rules can be written while relying on the tuple-space to abstract low level implementation and communication details.

In addition, LINC is highly reflexive. Because everything is a resource it is easy to control the execution of rules or group of rules. Coordination rules can be dynamically added, removed or moved to another object. Objects themselves can migrate and independently designed applications can be merged at run-time.

With all these properties and its reflexivity, LINC is a powerful environment to tackle the challenges of the future systems of systems.

We illustrated this with several real world case studies implemented on top of LINC. This demonstrated the ability of LINC to be used in contexts such as building automation, power efficiency or monitoring network systems.

Future work will focus on providing a high-level language such as automata in order to prove the correctness of the coordination and then to automatically generate the corresponding coordination rules. We envisage to use this mechanism for self-evolving system where we can prove that the evolution will not break the running application.

Acknowledgement. This work has been partially funded by the FP7 SCUBA project under grant nb 288079.

References

1. Jean-Marc Andreoli, Damián Arregui, François Pacull, and Jutta Willamowski. Resource-based scripting to stitch distributed components. In *Engineering and Deployment of Cooperative Information Systems*, pages 429–443. Springer, 2002.
2. Jean-Marc Andreoli, François Pacull, Daniele Pagani, and Remo Pareschi. Multiparty negotiation of dynamic distributed object services. *Science of Computer Programming*, 31(2):179–203, 1998.
3. BACNet. `http://www.bacnet.org/`, 2014.
4. Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, pages 17–44. Springer, 2001.
5. Jorge Barbosa, Fabiane Dillenburg, Gustavo Lermen, Alex Garzão, Cristiano Costa, and João Rosa. Towards a programming model for context-aware applications. *Computer Languages, Systems & Structures*, 38(3):199–213, 2012.
6. Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.
7. Manfred Bortenschlager, Gabriella Castelli, Alberto Rosi, and Franco Zambonelli. A context-sensitive infrastructure for coordinating agents in ubiquitous environments. *Multiagent and Grid Systems*, 5(1):1–18, 2009.
8. Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *Internet Computing, IEEE*, 4(4):26–35, 2000.
9. Thomas Cooper and Nancy Wogrin. *Rule-based Programming with OPS5*, volume 988. Morgan Kaufmann, 1988.
10. Laurent-Frederic Ducreux, Claire Guyon-Gardeux, Suzanne Lesecq, Francois Pacull, and Safietou Raby Thior. Resource-based middleware in the context of heterogeneous building automation systems. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 4847–4852. IEEE, 2012.
11. Tore Fjellheim, Stephen Milliner, Marlon Dumas, and Julien Vayssière. A process-based methodology for designing event-based mobile composite applications. *Data & Knowledge Engineering*, 61(1):6–22, 2007.
12. Chien-Liang Fok, Gruia-Catalin Roman, and Gregory Hackmann. A lightweight coordination middleware for mobile computing. In *Coordination Models and Languages*, pages 135–151. Springer, 2004.

13. David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.
14. Hoel Iris and Francois Pacull. Smart sensors and actuators: A question of discipline. *Sensors & Transducers Journal*, 18(special Issue jan 2013):14–23, 2013.
15. Christine Julien and G-C Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *Software Engineering, IEEE Transactions on*, 32(5):281–298, 2006.
16. KNX. `http://www.knx.org/`, 2014.
17. Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In *Coordination Models and Languages*, pages 121–135. Springer, 2013.
18. Edward A Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.
19. LONWorks. `http://www.lonmark.org/`, 2013.
20. Maxime Louvel and Francois Pacull. A coordinated matrix of rfid readers as interactions input. In *SENSORDEVICES 2013, The Fourth International Conference on Sensor Device Technologies and Applications*, pages 91–96, 2013.
21. Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Transactions on Software Engineering and Methodology*, 18(4):15, 2009.
22. Amy L Murphy, Gian Pietro Picco, and G-C Roman. Lime: A middleware for physical and logical mobility. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 524–533. IEEE, 2001.
23. Amy L Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, 2006.
24. Andrea Omicini and Mirko Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(01):53–59, 2011.
25. George A Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in computers*, 46:329–400, 1998.
26. Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.
27. SAPERE. `http://www.sapere-project.eu`, 2013.
28. SCUBA. `http://www.aws.cit.ie/scuba/`, 2011.
29. Scuba. Deliverable 5.3. `http://linc.middlewares.info/wiki1/images/8/81/Scuba_d_5_3.pdf`, 2013.
30. Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst.*, 6(2):14:1–14:24, June 2011.
31. Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Coordination Models and Languages*, pages 212–229. Springer, 2012.
32. Günter von Bültzingsloewen, Arne Koschel, Peter C Lockemann, and H-D Walter. Eca functionality in a distributed environment. In *Active Rules in Database Systems*, pages 147–175. Springer, 1999.
33. Yongjie Zheng, Alvin TS Chan, and Grace Ngai. Mcl: a mobigate coordination language for highly adaptive and reconfigurable mobile middleware. *Software: Practice and Experience*, 36(11-12):1355–1380, 2006.